Optimized Distribution of Synchronous Programs via a Polychronous Model

Ke Sun, Loïc Besnard, Thierry Gautier

Optimized Distribution of Synchronous Programs via a Polychronous Model

Ke Sun*, Loïc Besnard†, and Thierry Gautier*
*INRIA Rennes - Bretagne Atlantique, †CNRS/IRISA
*†Campus de Beaulieu, Rennes, France
Email: *{firstname}.{lastname}@inria.fr, †Loic.Besnard@irisa.fr

***Abstract*—This paper presents a distribution methodology for synchronous programs, applied in particular on programs written in the Quartz language. The given program is first transformed into an intermediate model of guarded actions. After user-specified partitioning, the generated sub-models are transformed into Signal processes. Using the multi-clock calculation model of the Signal language, an optimized data-flow network can be automatically constructed. The optimization includes reducing the communication quantity and the computation load, with no change to the interface behaviors.**

I. Introduction

Synchronous programming languages such as Esterel [1], Lustre [2] or Quartz [3] are all based on the *synchronous hypothesis* [4]. Under this assumption, system behaviors are projected onto a discrete sequence of logical instants. As the sequence is discrete, nothing occurs between two consecutive instants. Such temporal abstraction can greatly facilitate safety-critical reactive system design. It enforces deterministic concurrency of the system: Heisenbugs (i.e., bugs that disappear when one tries to simulate/test the system) are avoided; the system behaviors become predictable. It is also the key to a straightforward translation of synchronous programs to hardware circuits [5]. Furthermore, it guarantees semantic precision by using mathematical models, such as Mealy machine, as supporting foundations. These models enable a series of efficient optimization, compilation and verification techniques to be applied on synchronous programs.

Synchronous programs are modeled on centralized architectures with zero-time communications: any signal emitted by some component is instantaneously received by others. However, in real world, safety-critical reactive systems [6] [7] practically operate over distributed architectures with delayed communications. This mismatch results in a wide gap between centralized design model and distributed implementation. To get over this, *desynchronization* is introduced with the following requirement: how to distribute a synchronous program while preserving the same observable behaviors, i.e., the same interface behaviors. This issue has drawn a considerable attention both in theoretical challenges and in industrial relevance.

For Esterel and Lustre, a series of distribution methods have been proposed [8]. In [9], starting from a synchronous program, a network of communicating *codesign finite state machines* is constructed. The distribution of synchronous programs via the modeling as *finite deterministic automata* [10] is presented in [11]. Based on it, the extended method in [12] focused on the automatic deduction of distributed systems from centralized synchronous circuits. Furthermore, how to distribute synchronous programs to fulfill temporal constraints is introduced in [13]. The primary goal of these distribution methods does not include an optimization of the communication and the computation, which is in the focus of this paper.

A related distribution methodology is presented in [14]. Its procedure consists of three steps: 1) the given synchronous program is compiled into an intermediate AIF model [15], which is a common intermediate format for various synchronous languages [16] [17]; 2) the data dependencies within the model are analyzed and represented in a dependency graph, then the generated graph is subsequently partitioned into sub-graphs in which partitions can be made horizontal (for a pipelined execution) or vertical (for a parallel execution); 3) a synchronous elastic system [18] is synthesized by generating a distributed component for each sub-graph and establishing a communication infrastructure with synchronous elastic flow (SELF) protocol [18].

Owing to the analogy to synchronous hardware circuit, one can assume that the given synchronous program holds a *master clock* driving the whole program computation. We refer to such program as a *mono-clocked* program [19]. Furthermore, it follows the model of computation (MoC) *strict synchrony*: during each instant, each input channel always reads data and each output channel always produces data. This feature may cause unnecessary communications or computations. In [20], the communication quantity is reduced, according to an evaluation of communication necessity: a producer transmits a value only when it is really required for the calculation of a receiver. However, it may result in highly overestimated computations for evaluating communication necessity. To avoid this, the evaluation can be refined by constructing additional communications, while this operation may augment the communication. An optimization method on computations is presented in [21]: a condition is computed for every variable to determine whether its value is required for current or future computations; when the condition does not hold, the computation of the corresponding variable can be suppressed.

A new optimized distribution methodology is proposed, which is based on a more flexible MoC. In contrast to synchronous languages, the polychronous language Signal [22] [23] [24] is based on the MoC *polychrony*[1]. As its name suggests, a polychronous program makes use of multiple clocks to drive its execution, the system behaviors are defined on a partially ordered set of instants. One can consider that each component in the program holds its own master clock, and there is no longer a master clock for the whole program.

[1]From the Greek "poly chronos" to mean multiple clocks.

We refer to such program as a *multi-clocked* program. If there is no relation between the master clocks, the *multi-clocked* program does not follow a linear timeline, but a nonlinear one. Nonlinear instants t_1, t_2 are such that there is no order relation between them. The system behaviors over nonlinear instants occur asynchronously. Hence, the Signal language allows one to conveniently model the asynchronous communications between synchronous modules. This exactly accords with the requirement of distributed systems. In addition, nonlinear timeline implies the possibility to avoid unnecessary synchronization, thereby enabling potential optimization [25]. Due to these advantages, Signal is particularly suited as a coordination layer to finally build a data-flow network over desynchronized processing locations.

Based on the multi-clock calculation model of the Signal language, we propose a distribution methodology for the synchronous programs. First, the given synchronous program is compiled into AIF model. Second, according to the user specifications, the generated model is partitioned into sub-models. After the partitioning, the generated sub-models are transformed into equivalent Signal processes. Then, the unnecessary constraints are eliminated from the processes to avoid unnecessary synchronization. Finally, within the Signal framework, the minimal frequencies of communication and computation are computed via multi-clock calculation. This operation can efficiently reduce the communication quantity and the computation load. Along this way, an optimized data-flow network over desynchronized processing locations can be constructed. Note that both the methodology in [14] and ours process AIF model. Both methods are thereby independent of particular synchronous languages.

The rest of this paper is organized as follows. Section II briefly reviews the intermediate AIF model, which serves as the starting point for the distribution procedure, and introduces the Signal language and its associated polychronous semantic model. Section III presents the user-specified model partitioning and the transformation from partitioned models to Signal processes. In Section IV, how to achieve the optimization both on communications and on computations is presented in detail. Meanwhile, our methodology is illustrated through case studies in Section V. Finally, we conclude this paper and look forward the perspectives.

II. FOUNDATIONS

A. AIF Model & Synchronous Guarded Actions

To process synchronous programs, it is quite natural to compile them into intermediate models at first. In this way, the whole processing can be modularly divided into several steps and the models can be reused for different purposes [8], such as validation, comparison, model transformation and code generation. Furthermore, the processing on the intermediate model is independent of particular synchronous languages. The distribution of synchronous programs converts to two steps: 1) compilation into intermediate models; 2) distribution of intermediate models. The intermediate model devoted to the Quartz language is the Averest Intermediate Format (AIF) model [15]. The compilation of Quartz programs into AIF models has been implemented in the Quartz/Averest framework[2] [26]. The essential part of AIF model is AIF model behaviors that are described by a set of synchronous guarded actions.

[2]http://www.averest.org

Guarded actions are designed in the spirit of traditional guarded commands [27], which are well-established intermediate code for the description of concurrent systems. The guarded actions are in the form of $\gamma \Rightarrow \mathcal{A}$, where the Boolean condition γ is called *guard* and $\mathcal{A}$ is called *action*. The guards represent the complex control structure of the synchronous program. The actions represent the assignments to variables. Due to the category of assignments, each guarded action has either the form $\gamma \Rightarrow x = \tau$ (called guarded immediate action) or $\gamma \Rightarrow next(x) = \tau$ (called guarded delayed action). Once the guard is evaluated to *true*, the corresponding action instantaneously starts. Both kinds of assignments evaluate the expression τ at the current instant. Then, the immediate assignment $x = \tau$ instantaneously transfers the value of τ to the variable x, whereas the assignment effect of the delayed one $next(x) = \tau$ takes place at the next instant. When the value of a variable x cannot be determined by any action, its value is determined by the *absence reaction*. It determines the value, according to the storage type of the assigned variable: a *non-memorized* variable is reset to the default value (denoted as def_value, the particular value depends on the data type); a *memorized* variable keeps its previous value, or takes its default value at the initial instant.

Guarded actions describe model behaviors via two parts: the *data-flow* part computes locals and outputs; the *control-flow* part computes *labels*, which denote the pause locations of control-flow. Guarded actions in the control-flow part are in the form of $\gamma \Rightarrow next(l) = true$, where label l is a non-memorized Boolean local denoting a pause location. If l holds at the current instant, it means that the control-flow reached the pause location of l at the end of the previous instant, then it resumes from this location at the beginning of the current instant. Note that more than one label can hold at the same instant. This enables the description of the parallelism feature [3] of synchronous programs.

B. The Signal Language

In Section II-B1, we introduce the polychronous model as the formal basis of Signal. Then, an overview of the language is given in Section II-B2.

1) Polychronous Model: We start with the following sets: $\mathbb{X}$ is a countable set of variables; $\mathcal{B} = \{ff, tt\}$ is a set of Boolean data values where *ff* and *tt* respectively denote *false* and *true*; $\mathcal{V}$ is a non-empty set of data values, $\mathcal{B} \subset \mathcal{V}$; and $\mathbb{T}$ is a dense set equipped with a partial order relation, denoted by $\leq$. The elements in $\mathbb{T}$ are called *tags*. We now introduce the notion of *time domain*.

Definition 1 (time domain)**.** *A time domain is a partially ordered set $(\mathcal{T}, \leq)$ where $\mathcal{T} \subset \mathbb{T}$ that satisfies: $\mathcal{T}$ is countable; $\mathcal{T}$ has a lower bound $0_\mathcal{T}$ for $\leq$, i.e., $\forall t \in \mathcal{T}, 0_\mathcal{T} \leq t$; $\leq$ over $\mathcal{T}$ is well-founded; the width of $(\mathcal{T}, \leq)$ is finite.*

$(\mathbb{T}, \leq)$ provides a continuous time dimension. $(\mathcal{T}, \leq)$ defines a discrete time dimension that corresponds to the logical instants [24], at which the presence/absence of data can be observed during the system execution. Thus, the mapping of $\mathcal{T}$ on $\mathbb{T}$ allows one to move from "abstract" descriptions to "concrete" descriptions [28].

A *chain* $(C, \leq) \subseteq (\mathcal{T}, \leq)$ is a totally ordered set of tags that admits a lower bound 0_C. The notation $t + 1$ means the immediate successor of a tag t in C, which satisfies $\forall t' \in C, t \leq t' \Rightarrow t + 1 \leq t'$. We denote the set of all chains in $\mathcal{T}$

by $\mathcal{C}_\mathcal{T}$.

Definition 2 (event)**.** *An event on a given time domain $\mathcal{T}$ is a pair $(t, v) \in \mathcal{T} \times \mathcal{V}$, which associates a tag t with a data value v.*

All the events whose tags belong to the same chain, can constitute a data-flow. Formally,

Definition 3 (signal)**.** *A signal $s : C \to \mathcal{V}$ is a function from a chain of tags to a non-empty set of data values, where $C \in \mathcal{C}_\mathcal{T}$. The domain of s is denoted by $tags(s)$.*

Two signals s_1 and s_2 are identical, denoted by $s_1 = s_2$, if and only if $tags(s_1) = tags(s_2)$ and $\forall t \in tags(s_1), s_1(t) = s_2(t)$. $\mathcal{S} = \cup_{C \in \mathcal{C}_\mathcal{T}}(s : C \to \mathcal{V})$ is the set of signals over the time domain $(\mathcal{T}, \leq)$.

Definition 4 (behavior)**.** *Given a finite subset $\mathcal{X}$ of a countable set $\mathbb{X}$ of variables, a behavior over $\mathcal{X}$ is an injective function $b : \mathcal{X} \to \mathcal{S}$. The domain of b is denoted by $vars(b)$.*

The restriction of b over a set of variables X, denoted by $b|_X$, is the behavior defined by $vars(b|_X) = X \cap vars(b)$ and $\forall x \in vars(b|_X), (b|_X)(x) = b(x)$.

Two behaviors b_1 and b_2 are compatible, denoted by $b_1 \updownarrow b_2$, if and only if $b_1|_{vars(b_2)} = b_2|_{vars(b_1)}$. The composition of two compatible behaviors $b_1 : \mathcal{X}_1 \to \mathcal{S}$, $b_2 : \mathcal{X}_2 \to \mathcal{S}$ is a behavior $(b_1|b_2) : (\mathcal{X}_1 \cup \mathcal{X}_2) \to \mathcal{S}$ defined by $\forall x \in \mathcal{X}_1, (b_1|b_2)(x) = b_1(x)$ and $\forall x \in \mathcal{X}_2, (b_1|b_2)(x) = b_2(x)$.

Definition 5 (process)**.** *A process p is a set of behaviors defined over the same domain. The domain is denoted by $vars(p)$, which satisfies $\forall b \in p, vars(b) = vars(p)$.*

Definition 6 (composition of processes)**.** *The composition of two processes p_1 and p_2, denoted as $p_1|p_2$, is a process consisting of the compositions of any two compatible behaviors $b_1 \in p_1$ and $b_2 \in p_2$:*

$$p_1|p_2 = \{(b_1|b_2)|(b_1, b_2) \in p_1 \times p_2, b_1 \updownarrow b_2\}.$$

The notions presented in this section are sufficient to express the semantics of Signal elementary concepts within this polychronous model [24]. In the remainder of this paper, we also use them to formulate the optimization scheme.

2) An Overview of the Signal Language: Signal is a polychronous language processing unbounded series of typed values, called *signals*. At any tag t, a signal x (corresponding to $b(x)$ in the polychronous model, where b is a behavior) may be present or absent: when present (i.e., $t \in tags(b(x))$), it holds some value; when absent (i.e., $t \notin tags(b(x))$), it holds no value.

The presence status of x is denoted by its associated clock $\hat{x} : tags(b(x)) \to \{tt\}$. Furthermore, the Signal language supports clock calculation. The basic operations contain clock union $x_1 \hat{+} x_2$: $tags(b(x_1)) \cup tags(b(x_2)) \to \{tt\}$; clock intersection $x_1 \hat{*} x_2$: $tags(b(x_1)) \cap tags(b(x_2)) \to \{tt\}$; and clock difference $x_1 \hat{-} x_2$: $tags(b(x_1)) \backslash tags(b(x_2)) \to \{tt\}$. In order to enable reasoning on clock calculation, we define $\hat{0}$ for the empty clock (i.e., $\hat{0} : \emptyset \to \{tt\}$) and $[x]$ (resp. $[\neg x]$) for the clock at which tags a Boolean signal x holds the value *true* (resp. *false*).

a) Declarative Constraints

A Signal program declares constraints on the involved signals, that must be satisfied by both values and clocks. The constraints on clocks are referred to as *clock relations*, including synchronization relation $x_1 \hat{=} x_2$, i.e., $tags(b(x_1)) = tags(b(x_2))$; inclusion relation $x_1 \hat{\leq} x_2$, i.e., $tags(b(x_1)) \subseteq tags(b(x_2))$; and mutual exclusion relation $x_1 \hat{\#} x_2$, i.e., $tags(b(x_1)) \cap tags(b(x_2)) = \emptyset$.

Besides the explicit clock relations, constraints can be implicitly declared by the Signal equations. Each equation is a definition associating one defined signal with a Signal expression built on operators over signals. The operands of the operators can be signals or expressions.

There are two kinds of definitions:

- A *complete definition* $(:=)$ is an equation in which the defined signal is assigned with the associated expression. For instance, $y := x$ is a complete definition of y that is assigned with x when x is present; y is synchronized with x:
$$\forall t \in tags(b(x)), b(y)(t) = b(x)(t) \text{ and } y \hat{=} x.$$
- A *partial definition* $(::=)$ is an equation in which the defined signal is assigned with the associated expression when the expression is present. For instance, $y ::= x$ is a partial definition of y that is assigned with x when x is present; when x is not present, the value of y depends on other partial definitions:
$$\forall t \in tags(b(x)), b(y)(t) = b(x)(t).$$

According to the implied clock relations, the Signal expressions can be classified into two families: *synchronous expressions* (i.e., all the involved signals have the same clock) and *polychronous expressions* (i.e., the involved signals may have different clocks). The primitive expressions include

- *Instantaneous function*: $x := f(x_1, \ldots, x_n)$ defines a point-wise n-ary function on sequences of values, in which all the signals $x, x_1, \ldots, x_n$ are synchronous.
- *Delay*: $x := x'\$1$ **init** def_value defines that x and x' are synchronous; the current value of x is equal to the previous value of x' and equal to the default value def_value at the initial tag.
  - $x \hat{=} x'$
  - $b(x)(0_{\mathcal{C}_x}) = def\_value$
  - $\forall t \in tags(b(x)), 0_{\mathcal{C}_x} < t + 1 \Rightarrow b(x)(t + 1) = b(x')(t)$

  where $\mathcal{C}_x = tags(b(x))$.
- *Downsampling*: $x := x'$ **when** b defines a downsampling of the signal x' that occurs only when both $\hat{x'}$ and the *downsampling condition* b hold *true*.
  - $x \hat{=} x' \hat{*} [b]$, where $[b] \hat{=}$ **when** b
  - $\forall t \in tags(b(x)), b(x)(t) = b(x')(t)$
- *Deterministic merging*: $x := x_1$ **default** x_2 defines that the clock of x is the clock union of x_1 and x_2, its value is equal to x_1 when x_1 is present, otherwise equal to x_2 when x_1 is absent but x_2 is present.
  - $x \hat{=} x_1 \hat{+} x_2$
  - $\forall t \in tags(b(x_1)), b(x)(t) = b(x_1)(t)$
  - $\forall t \in tags(b(x_2)) \backslash tags(b(x_1)),$ $b(x)(t) = b(x_2)(t)$
- *Completion*: $x ::=$ **defaultvalue** x' completes the definition of x. Given the partial definitions $x ::= x_1, x ::= x_2$, where $x_1 \hat{\#} x_2$, x is then identical to x' when x_1 and x_2 are absent but x and x' are present.